Bicoloring

To check whether a graph can be colored using only two colors, we can perform a Breadth-First Search (BFS) and try to assign colors in a consistent way. Since the graph is connected, we may begin the traversal from node 0. Every node is initially marked as “uncolored” (for example, using –1). We then give the starting node the first color (say 0) and place it in a queue to start the BFS.

During the BFS, we repeatedly take a node from the front of the queue and examine all of its neighbors.

- If a neighbor hasn’t been colored yet, we assign it the opposite color of the current node and push it into the queue.
- If a neighbor already has a color, we compare it with the current node’s color.
    - If both colors match, it means two connected nodes need the same color, which breaks the rules of two-coloring. That immediately tells us the graph cannot be bicolored.
    - If their colors differ, we continue without issues.

If the BFS completes without encountering any color conflicts, then every node received one of the two colors in a valid way—meaning the graph is bicolorable.
This BFS method is simple, efficient, and correctly detects the presence of odd cycles, which are the reason a graph fails to be bipartite.

BFS Procedure for Bicoloring

1. Mark all nodes as uncolored

Imagine every node starts in a “no color” state.
This helps us identify which nodes still need a color assignment during BFS.

2. Begin the BFS

Choose a starting point—typically node 0.
Give it the first color (color 0) and push it into a queue.
This queue keeps track of which nodes should be processed next.

3. Process nodes from the queue

While the queue still contains nodes:

- Take the front node and call it the current node.

- Look at all its adjacent nodes (neighbors).

4. Coloring the neighbors

For each neighbor:

- If the neighbor is uncolored:
    - Assign the opposite color of the current node.
    - Push the neighbor into the queue so its neighbors can also be checked later.
- If the neighbor already has a color:
    - Compare the colors.
    - If the neighbor and current node share the same color, a conflict appears.
        - This means the graph cannot be properly colored with just two colors.
        - At this moment we conclude the graph is not bicolorable.
    - If their colors differ, BFS continues normally.

5. Continue until BFS finishes

BFS explores the graph layer by layer.
Nodes in one layer end up sharing the same color, and the next layer receives the opposite color.
If the queue becomes empty without detecting any conflict, the two-coloring is valid.

6. Final Decision

- No conflicts found: The entire graph was colored using two colors → bicolorable.
- Conflict detected: Some adjacent nodes needed the same color → not bicolorable.

Input:

3

3

0 1

1 2

2 0

Stepwise BFS Coloring

Initialization

• All nodes are uncolored: o Node 0 → uncolored o

Node 1 → uncolored o Node 2 → uncolored

• BFS queue is empty.

Step 1: Start BFS from node 0

• Assign color 0 to node 0.

• Add node 0 to the BFS queue.

Colors:

• Node 0 → 0

• Node 1 → uncolored

• Node 2 → uncolored

Queue: [0]

Step 2: Process node 0

• Neighbors of node 0: 1 and 2

• Both are uncolored → assign opposite color (1)

• Add neighbors to queue

Colors:

• Node 0 → 0 • Node 1 → 1

• Node 2 → 1

Queue: [1, 2]

Step 3: Process node 1

• Neighbors of node 1: 0 and 2

• Neighbor 0 → color 0 → fine

• Neighbor 2 → color 1 → conflict detected o Both node 1 and node 2 are connected and have the same color o Conflict means the graph cannot be bicolored

Step 4: Stop BFS

• BFS stops immediately because a conflict was found

• Graph contains an odd cycle (triangle) → impossible to 2-color

Output:

Not Bicolored.

Pseudocode:

```
LOOP forever:
   INPUT n
   IF n equals 0:
      EXIT loop

   INPUT m

   CREATE graph with n empty lists

   FOR each of the m edges:
      INPUT x, y
      APPEND y to graph[x]
      APPEND x to graph[y]
```

```
SET color array of length n to all -1

INITIALIZE an empty queue q
SET color[0] to 0
PUSH 0 into q

bipartite = true

WHILE q is not empty AND bipartite is true:
    curr = POP from q

    FOR every node next in graph[curr]:
        IF color[next] == -1:
            color[next] = 1 - color[curr]
            PUSH next into q
        ELSE IF color[next] == color[curr]:
            bipartite = false
            BREAK

IF bipartite is true:
    PRINT "BICOLORABLE."
ELSE:
    PRINT "NOT BICOLORABLE."
```

Code: https://github.com/pulokdas062/Graph-Algorithm/edit/main/BFS/BICOLORING/bicolor.cpp